Report No. UIUCDCS-R-72-503

SIMULATION OF A TREE PROCESSOR

by

Larry Allen Swanson

January, 1972

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-72-503

Simulation of a Tree Processor

by

Larry Allen Swanson

January, 1972

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by US NSF GJ 27446 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, January, 1972

TABLE OF CONTENTS

ACKNOWLEDGEMENT

The author wishes to express appreciation to Professor D. J. Kuck for valuable assistance and utmost patience during the development and writing of this thesis. The author acknowledges Paul Budnik and Yoichi Muraoka who designed and developed parts of the simulator. The simulator's input was produced by a program written by Joseph Han. Funds from National Science Foundation Grant USNSF-GJ-2744 administered by Professor Kuck were used for computer time to complete part of the simulator. Finally, the author appreciates the many hours spent by Diana Mercer and the author's wife, Judith, typing the manuscript.

1. INTRODUCTION

To increase the throughput of data in computers, the organization of several computing machines have been designed to take advantage of parallelism in computing. Examples of such machines are ILLIAC IV, the Burroughs 5500, IBM 360/91 and the Control Data Star. A design which seems to exploit the natural structure of assignment statements has been proposed by Professor David Kuck.[1] This paper will discuss part of the motivation of such a machine, the overall organization of Kuck's proposal and describe a timing simulator which was written to simulate a class of such machines. In addition, the experiments which were run on the simulator and their results will be discussed.

2. MOTIVATION FOR A TREE PROCESSOR

Consider the assignment statement R=B*C+D*E which maps on to the binary tree as shown in Figure 1. As

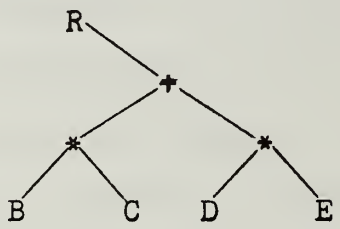


Figure 1

proposed by Kuck, a natural connection of processors in a parallel processor would be in a tree configuration as shown in Figure 2 for the example assignment statement. Thus, PE2

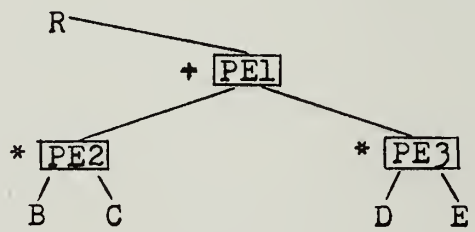


Figure 2

and PE3 would calculate the products of B and C and D and E simultaneously. These intermediate results would then be routed to PE1 where the addition could take place. The statement would be processed in two steps whereas three steps are required by a single processor. Obviously, another similar assignment statement

could be started in the processors PE2 and PE3 while PE1 is processing the addition of the intermediate results. Thus, if four assignment statements similar to the given example were to be processed, twelve steps would be needed by a single processor while only five steps would be required in the tree processor. For ideally distributed operands and operators, one statement of N operands requires N-1 processing steps in a single processor but only $\log_2 N$ steps for a tree processor with N-1 processors. More significant is the fact that if S such statements of N operands are to be processed, only $S+\log_2 N-1$ steps are required for execution in a N-1 processor tree whereas S(N-1) steps are needed by a single processor. Obviously, these expressions are for ideal situations and several problems of this design present themselves immediately.

One problem is that for real programs the distribution of operands and operators is much less than ideal. In addition, few assignment statement trees would fit onto any fixed size of a tree simulator perfectly; many assignment statement trees would be too small and few undoubtedly would be too large. Hence, buffers with a tag algorithm similar to that implemented in the IBM 360/91 would be required to drive the processors in the tree structure.

Another problem is supplying the tree processor with a sufficient amount of data so that it can be kept busy. Certainly at least one memory module for each processor in the bottom level of the tree processor would be necessary. However,

only in some very rare cases would the memory module associated with a processor in the bottom row contain the data required by the processor. Thus, a router or alignment network would be necessary to distribute data from the memory modules to the processor buffers. In addition, a router would be required to route data from the outputs of the processors to memory and also from the outputs of processors back to the bottom level of processors in the tree processor. Data is required to be routed from the outputs of processors to the input of bottom level processors whenever a result of one expression is required by a following expression. This condition will be referred to as "feedback".

The general machine organization discussed above is shown in Figure 3. (For simplicity, all descriptive diagrams of the tree processor will include only seven processors.) Since the performance of such a machine organization on real programs is extremely difficult to determine theoretically, it was decided that a timing simulator be written. The simulator's purpose is to help determine what type of routing network in each position would be required such that no bottlenecks occur in the overall system. In addition, the simulator will also help determine the buffer size necessary between processors and in the router networks such that the system contain no bottlenecks. Thus, throughput and buffer sizes will be measured for various machine configurations.

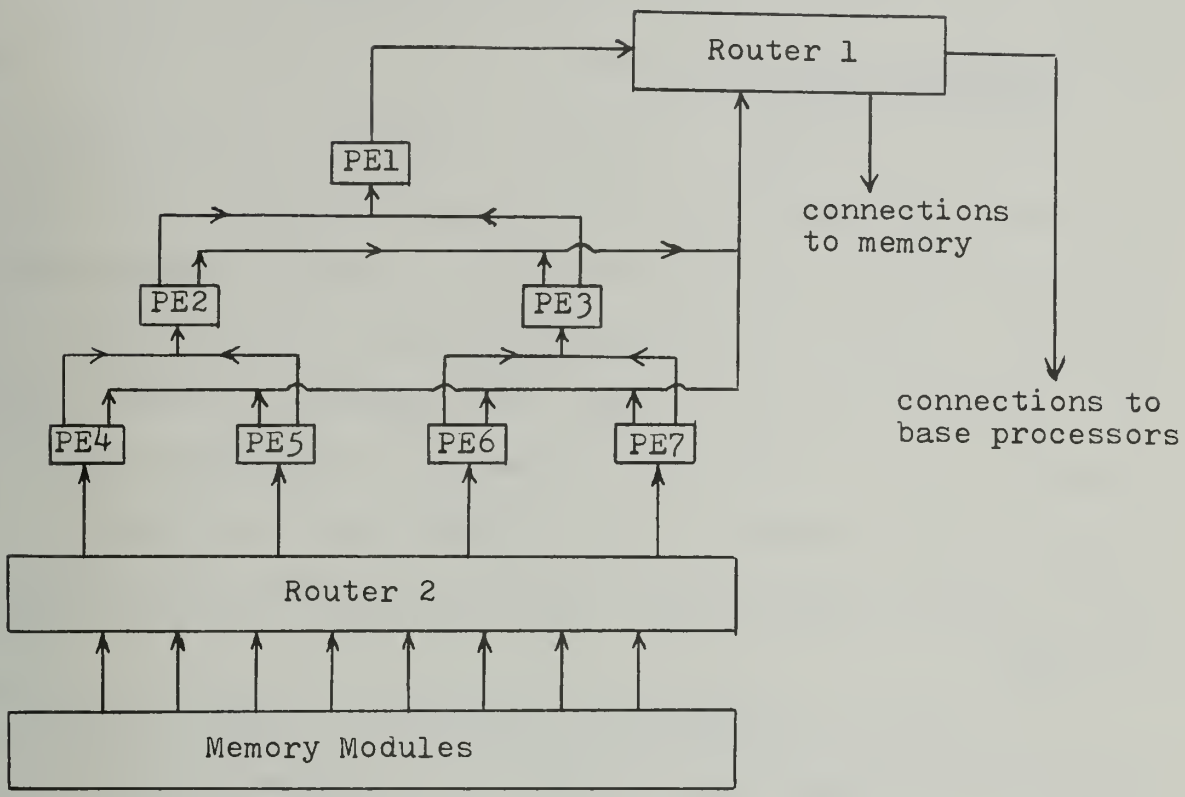

Router 1
PE1
connections
to memory
PE2
PE3
connections to
base processors
PE4
PE5
PE6
PE7
Router 2
Memory Modules

Figure 3

3. OVERALL DESCRIPTION OF SIMULATOR

3.1 Simulator Input

In an attempt to provide a parallel machine design which does not suffer from any serious bottlenecks, Kuck has proposed the series of elements in this parallel processor be pipelined, driven by buffers and that each element have an equal pipeline step size. Thus, the instruction decoder, tree processor and all routers are assumed to be pipelined with a common pipeline interval. This concept about the overall design should be kept in mind while reading the remainder of the paper.

Input to the simulator consists of height reduced assignment statement trees produced by a program written by Han[2] which implements some tree height reducing algorithms proposed by Muraoka[3]. A simple example is given here to give the flavor of the work done by Han's program. Consider the following FORTRAN assignment statement. As shown below in Figure 4,

$$R = A * (B+C*D) + E \qquad (1)$$

a four level, fifteen processor tree would be required to process the statement as written without using a temporary result and routing the result back to another processor as input. Note also that only four of the fifteen nodes in the tree processor would be utilized. (If other instructions were present in the

machine, the unscheduled processors might be able to do other useful work.) However, with a distribution and a proper

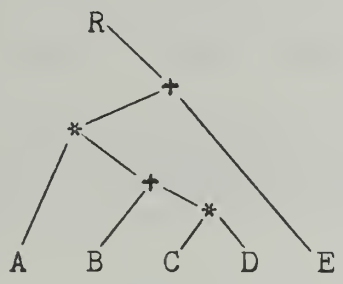


Figure 4

permutation of the input variables, the expression can be evaluated with a three level tree and only a seven processor tree would be required to calculate the expression without feedback as shown in Figure 5.

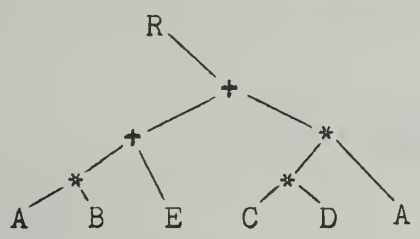


Figure 5

3.2 Simulator Scheduler

A scheduler designed and programmed by Paul Budnik drives the entire tree machine using the tree height reduced expressions generated by Han. The scheduler uses an instruction look ahead unit which can be varied in length for different

experiments. For any given program with a given number of processors in the processor tree, some instructions would not fill the tree while others would be too large to fit on the tree. Thus, the scheduler is designed to schedule more than one expression on the processor tree for each clock if one expression does not fill the entire processor tree during a particular simulation. The statements (2) and (3) would then

$$R1 = B * C + D + E \qquad (2)$$

$$R2 = G + H * I + J \qquad (3)$$

be scheduled to be input into a seven processor tree as shown in Figure 6. Note that the results R1 and R2 will emerge from processors PE2 and PE3 respectively.

The scheduler also cuts trees that are too large to fit on the processor tree during a given simulation by assigning

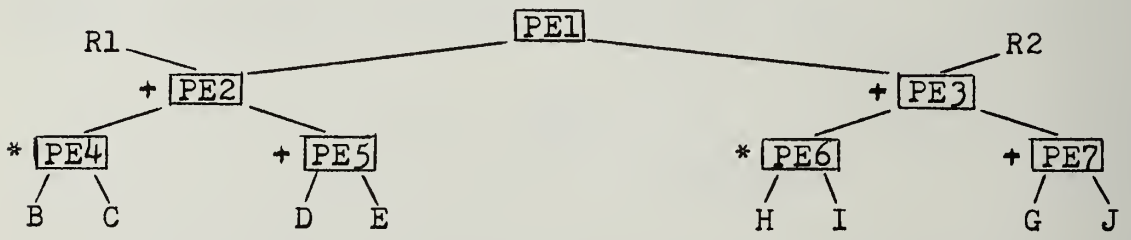


Figure 6

temporary variables to intermediate results and setting the proper tags to enable R1 to route the temporary result to the correct destinations. An example of such a case is shown on

the two trees in Figure 7 for statement (4). Note that a portion of another assignment statement tree

$$R = A * B * C + D * E * F + G * H + I \qquad (4)$$

or a complete assignment statement can be scheduled on processors PE6, PE7, and PE3 while the second part of the

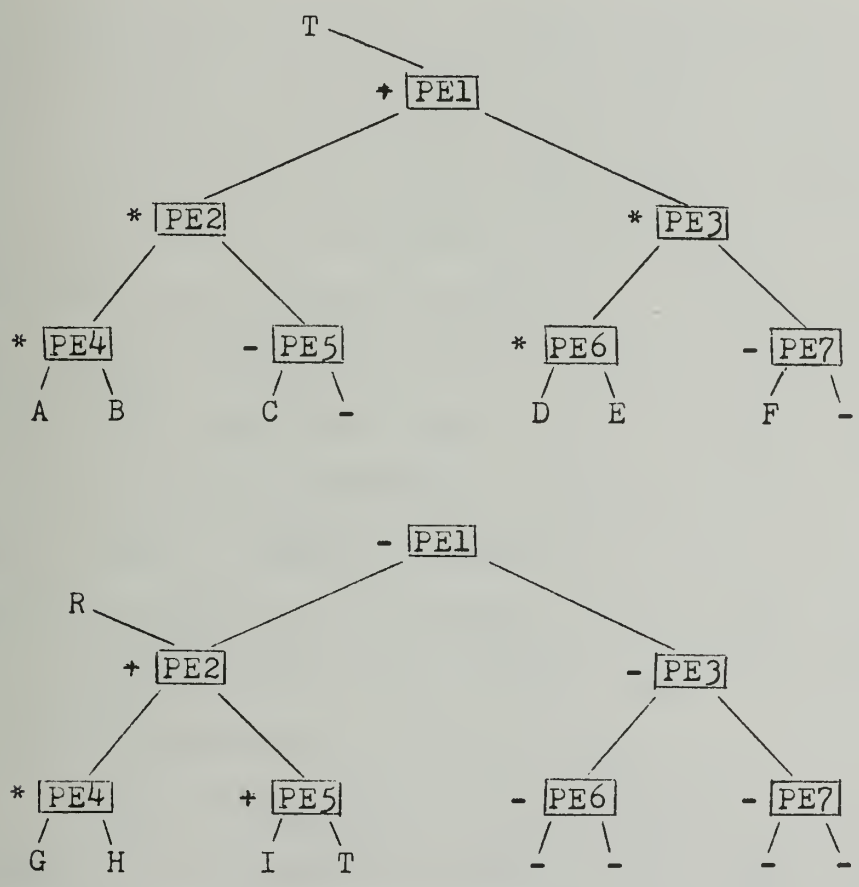


Figure 7

instruction is being processed. Notice also that PE5 and PE7 in the first tree only "pass along" their operands to the next

level. In Figure 7, - indicates either no operation or no input data.

Budnik's scheduler also handles dependency such as that in the instructions (5) and (6). The result of (5) is required as input data in (6). Thus, the scheduler was

$$R1 = A * B + C \qquad (5)$$

$$R2 = R1 + B * D \qquad (6)$$

designed to have the result of (5) routed back directly to the proper buffer in the bottom level of the processor tree instead of delaying instruction (6) until R1 arrived in memory. This is implemented by assigning a number in an increasing unary sequency starting from zero to each different variable which is encountered in the instruction stream. This number is the index of an array named MEML which indicates the location of the instruction producing the variable as a result in the machine. The number assigned to each variable as it is encountered in the instruction stream also indicates the memory module in which the variable will be assumed to be stored. The memory module number is the variable number modulo the number of memories being simulated. Thus, no fancy storage scheme was used for the simulator. The value of MEML indicates whether the variable is in memory, being processed as a left hand side in the machine before R1 or being processed as a left hand side in the machine inside R1.

Then if a variable is being produced as a result in the machine anywhere before R1 and the variable is required as input data to another instruction as in (5) and (6) above, another element is added to the linked list of destinations for the instruction producing the result. Thus, in our example another element would be added to the linked list of result destinations for instruction (5) so that R1 could be input to the proper processor buffer for instruction (6). INSTRORES, one of several arrays associated with the instruction format abbreviated by INSTR, is an array indexed by the instruction number which points to the head of a linked list of destinations to which the result of the instruction is to be routed. One of the elements in the linked list, INSTRFLG, indicates whether the result is to be routed to memory or back as input to a bottom level processor. Thus, if (5) and (6) were the only two instructions in the instruction stream, INSTRORES for (5) would point to a linked list containing two entries. One entry would indicate a destination to the memory location of R1 and the other entry would indicate a processor destination for instruction (6).

The destinations for the results going to memory or to a processor as input are stored in the array INSTRDATA. If the INSTRFLG entry in the linked list of parallel arrays indicates that the result should be routed as input to a processor, another parallel array INSTRST indicates the target instruction for the feedback result. Feedback results occur

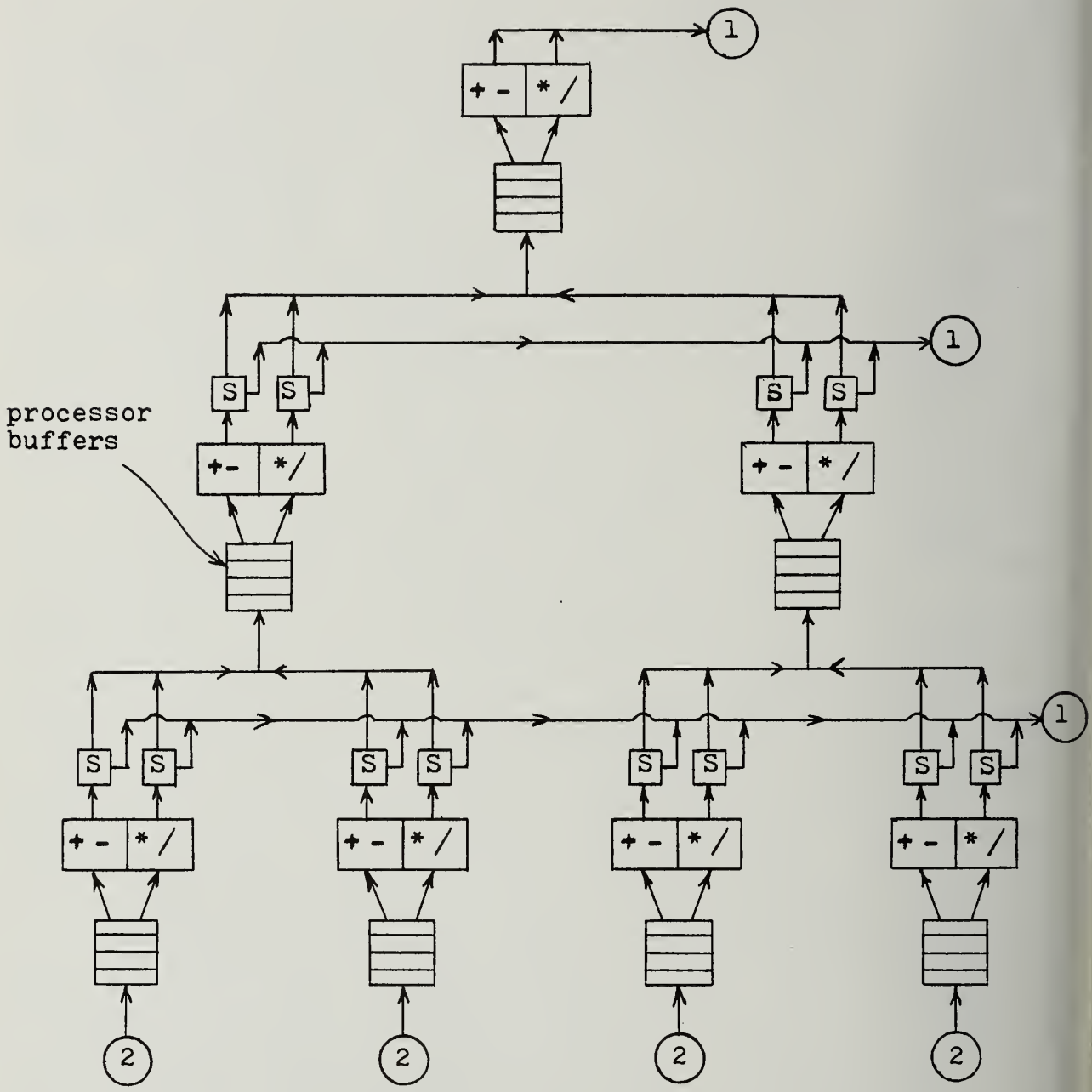
when assignment statement trees are cut by the scheduler producing temporary results or when dependency occurs as in the above example. The parallel arrays INSTRDATA, INSTRFLG and INSTRST are linked together by another equally dimensioned array named INSTRLINK.

Another array parallel to INSTRORES and of the length of the look ahead is INSTRTYP. This array differentiates between assignment statements of the type variable = variable and those of the type variable = expression. Since it is obviously inefficient for the tree processor to handle any statement of the type variable = variable, INSTRTYP is tagged to indicate that the instruction is of this type and the instruction's linked list of results (pointed at by INSTRORES) indicates to which destinations the left hand side is to be routed. The interfacing between the scheduler and R2 then insures that an instruction of this type does not reach the processing tree but that each of the elements in the linked list of result destinations is placed into the proper routers. Summarizing, it would be well to point out again that INSTRORES and INSTRYP are arrays associated with the look ahead unit and are of the same length. INSTRDATA, INSTRST, INSTRLINK and INSTRFLG are four parallel arrays which provide proper information to the simulator regarding what is to be done with the left hand side of an instruction when it emerges from the processor tree or when it is available from memory and is tagged of the type variable = variable.

3.3 Processor Simulator

The design and programming of the tree processor itself was done mostly by Yoichi Muraoka. Its general structure is shown in Figure 8. The buffers between the processing elements are shown explicitly in the diagram. The length of these processor buffers can be varied between simulation runs. Each of the processors contains an add/subtract unit as well as a multiply/divide unit as shown. Both units are assumed to be pipelined and the length of the pipeline in either the adder or multiplier can be varied between each simulation. The number of operands that each processor can accept and the number of outputs from each processor per clock is also a parameter to the simulator. Since both an adder and multiplier are present and are independent in each processor, the system can be allowed to permit four operands to arrive at each processor each clock - two for the adder and two for the multiplier. If four operands are input to each processor tree node, two results must be permitted to leave the processor during a particular clock. Another option provides that an operand can be "routed around" a processor to the next level of the tree if no operation is to be performed on the data. An example of this case has been shown in Figure 7.

As mentioned previously, the need for the processor buffers arises from the fact that very few assignment trees found in real programs are both full and balanced. Here the term balanced indicates that each subtree of a particular



S indicates a switch determining whether output from a processor exits to the next level in the processor tree or to the result router.

1 indicates inputs to the result router (R1).

2 indicates inputs to the bottom level processors from the router between memory and the processors (R2).

Figure 8

processor take the same amount of time to execute and full indicates that all nodes of the processor would be scheduled for the tree processor. The following assignment statement can be scheduled on a full balanced tree as shown in Figure 9.

$$R = (A+B) * (C+D) + (E*F) + (G*H) \qquad (7)$$

However, in the schedule shown in Figure 6, the result from PE4 would not be calculated as soon as the result of the addition from PE5 since the multiply would take more processing time.

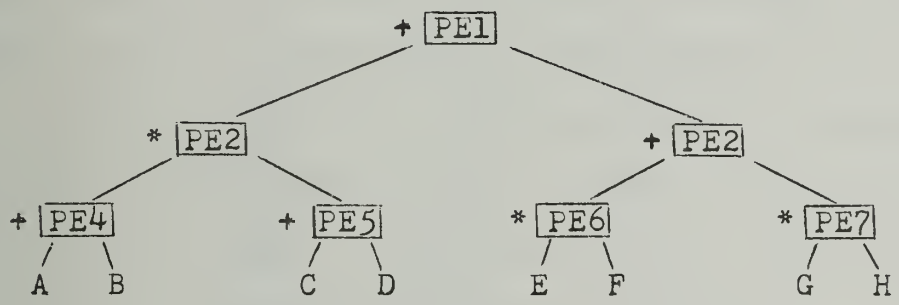


Figure 9

Hence, PE2 could not proceed until the product of B and C had been executed. Especially when sparse trees are being processed, it is not difficult to construct reasonable examples in which one whole instruction which originally was later in the instruction stream would emerge from the processor tree before the prior instruction. The inclusion of buffers before each processor in the processor tree certainly seems reasonable from an efficiency standpoint. Of course, the necessary length

of these buffers is one question which the simulator should help answer.

As mentioned previously in the discussion of the tree processor, the number of operands arriving at each processor per clock is variable. The delivery of a various number of operands at each processor per clock can be achieved two ways. Suppose one desires to have four operands arrive at each processor in each processor clock instead of two operands. First, the increase in the data rate can be achieved by increasing the speed of the memory such that each memory module can produce twice as much data in a given amount of time. Perhaps a more practical alternative is to provide twice as many memory modules to feed the same number of processors. Either scheme can be implemented in the simulator. The simulator will simulate situations in which 1, 2 or 4 memory modules are provided per processor in the bottom level. These rates correspond to 1/2, 1 or 2 arithmetic expressions arriving at the bottom row of the processors at each clock.

3.4 Machine Configurations

The simulator allows three general router configurations to be simulated. Figure 10 shows one of these configurations. For the discussion of the various machine configurations, the separate adder/subtract unit and multiply/divide unit in the processor will not be shown in an attempt to keep the diagrams fairly simple. In addition, dotted lines indicate the presence of more than one connection. In Figure 10 R2

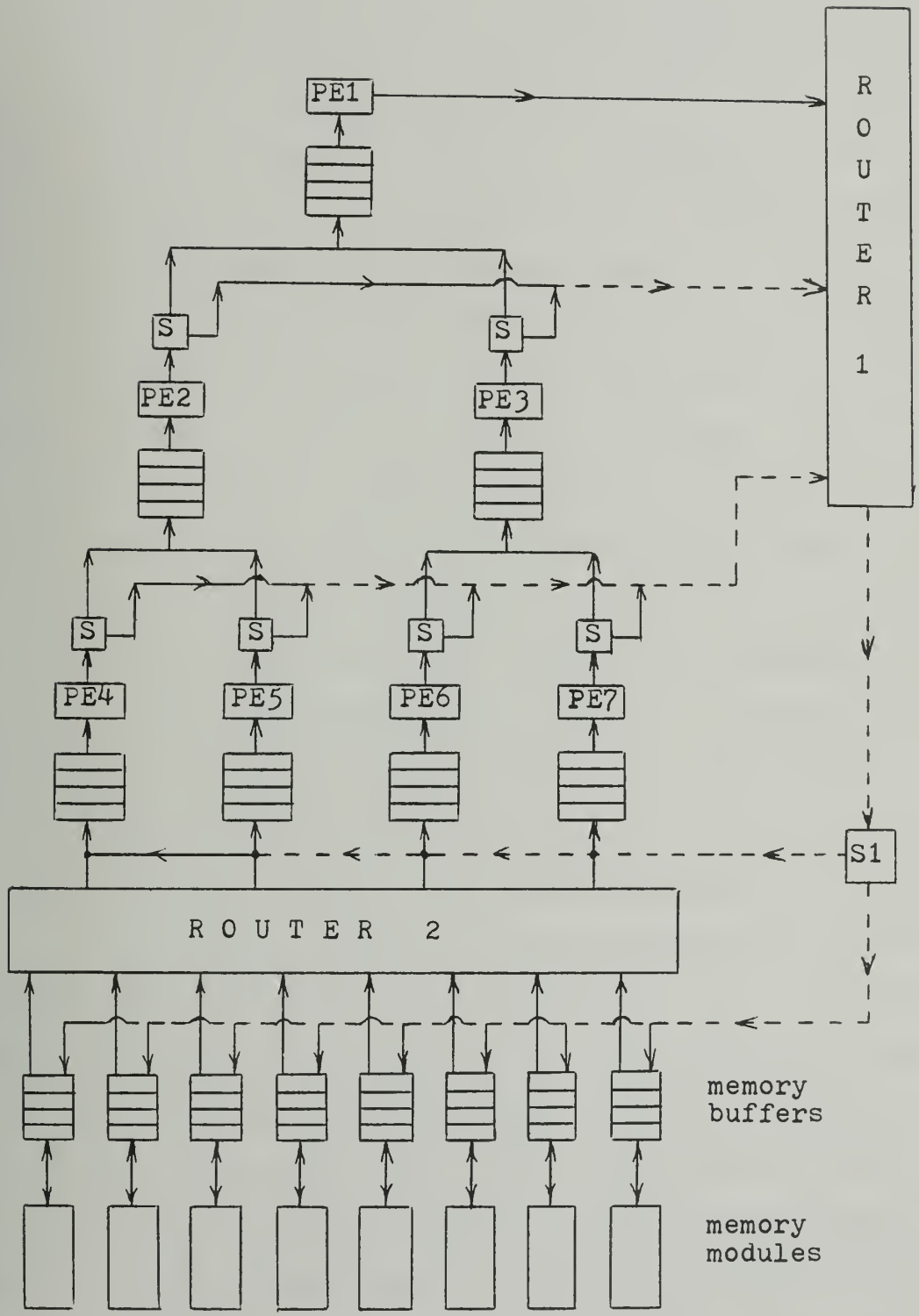

PE1
ROUTER 1
S
S
PE2
PE3
S
S
S
S
PE4
PE5
PE6
PE7
S1
ROUTER 2
memory buffers
memory modules

Figure 10

routes data from memory buffers to processor buffers in the bottom level of the tree processor. R1 routes data from the outputs of all processors to the inputs of the processors in the bottom level of the processor tree as well as data from the outputs of the processors to each of the memory modules. The square boxes marked S are switches which route results from each processor to the next processor or to the result router R1. Another set of switches labeled S1 represented by only one square box, are required to switch the output of R1 to the correct place - either to the input to one of the bottom level processors or back to the memory buffers. Notice that the number of switches of Type S1 is the number of processors in the bottom level of the processor tree. In Figure 10, the number of memories per processor shown is two; thus, one instruction should be entering the system per clock.

The other configuration with two routers is shown in Figure 11. In this case R2 routes data from memory buffers to the bottom level processors and also from the outputs of the processors in the processor tree to memory. In this configuration each piece of data in R2 would require a tag indicating whether the destination of the data is the processor buffers or the memory buffers. R1 only routes results from the processors back to the input of the bottom level processors. In other words, R1 only handles feedback results. In Figure 11 the switches labeled S determine whether results from processors should be passed to the next level of the tree or switched to be

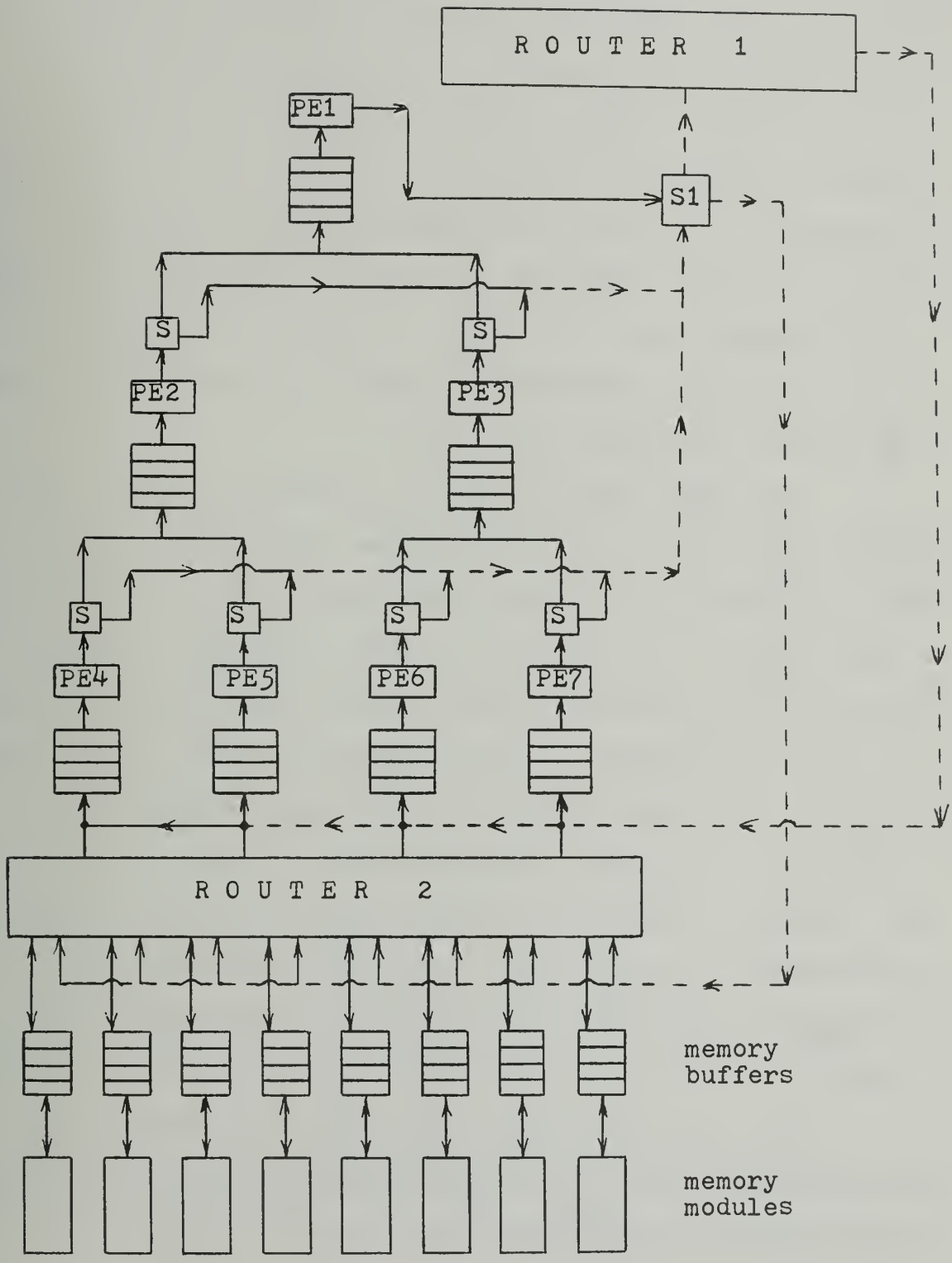

ROUTER 1
PE1
S1
S
S
PE2
PE3
S
S
S
S
PE4
PE5
PE6
PE7
ROUTER 2
memory buffers
memory modules

Figure 11

input to another switch represented by one block in the diagram labeled S1. The switches S1 determine whether the results from the processors are to be placed into R1 and subsequently be routed to one of the input buffers in the bottom level of processors or whether the results are placed into R2 and subsequently routed to one of the memory buffers. Notice the difference in the function of the switches labeled S1 in Figure 10 and those labeled S1 in Figure 11 and that the number of switches labeled S1 required in Figure 11 is equal to the number of processors in the processor tree.

The final general machine configuration uses three routers instead of two as in the former two configurations. Figure 12 shows that R1 routes data output from the processors to input buffers for the bottom level processors. R2 routes data only from the memory buffers to the input buffers for the bottom level processors. R3 routes data from the output of the processors to the memory buffers of the proper memory module for the data. Thus, each router performs a single specific task. Figure 12 closely resembles Figure 11 except that one of the outputs of switch box S1 ends at R3 instead of R2. Notice that in all three configurations the switches S are required.

Any of the following five types of routers can be placed in each or all of the routing network positions (R1, R2 and R3) during any simulation run:

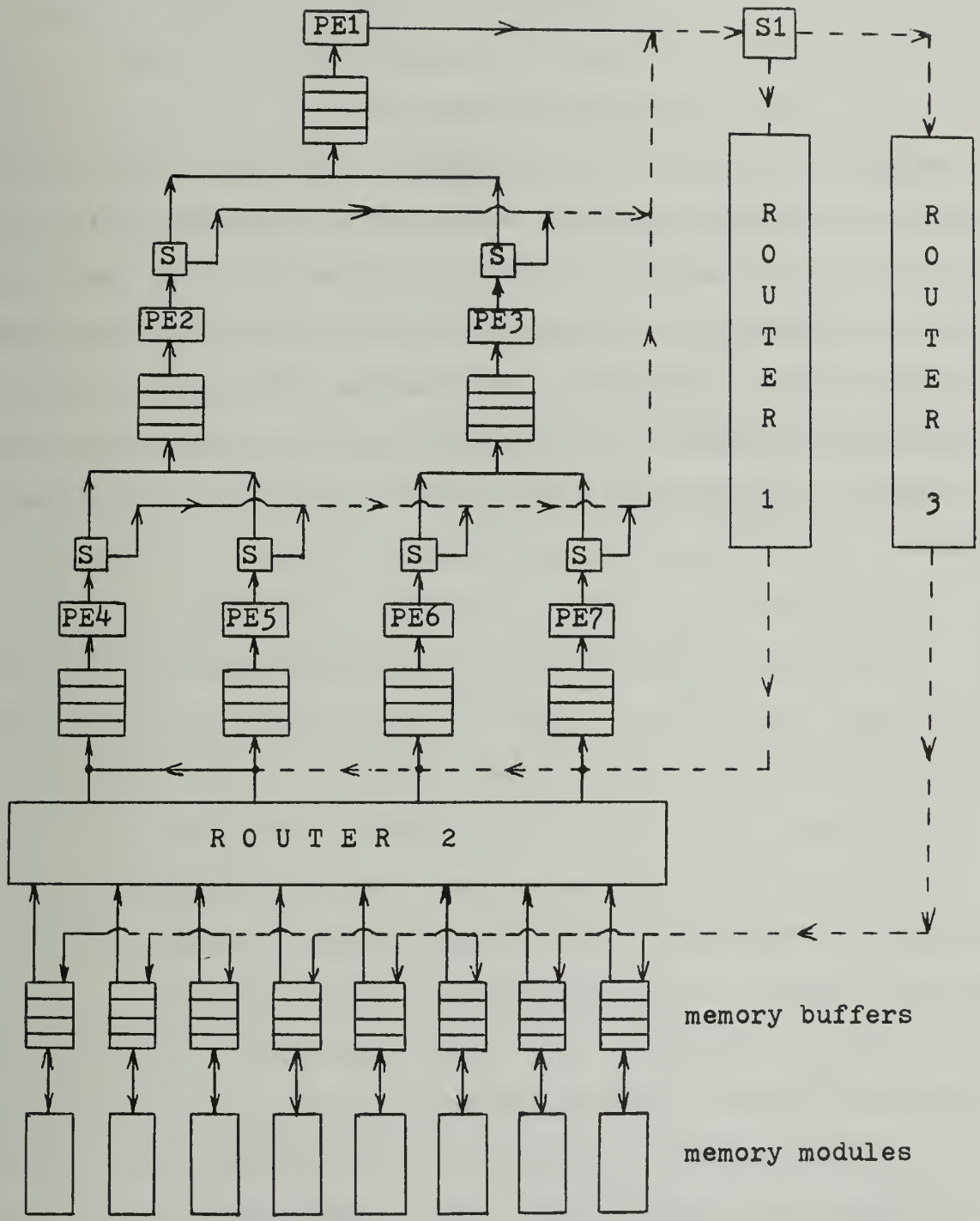

PE1
S1
S
S
PE2
PE3
R O U T E R 1
R O U T E R 3
S
S
S
S
PE4
PE5
PE6
PE7
R O U T E R 2
memory buffers
memory modules

Figure 12

Log Router[4]

Illiac IV Router

Semmelhaack Router

Batcher Network[5]

Crossbar Switch

In addition, the size of the processor tree (and therefore the size of the routers), memory buffer size, processor buffer size, number of memory modules, length of instruction look ahead and number of memory modules per processor in the bottom level are variable between simulations. The maximum number of following instructions to which the left hand side of an instruction can be linked is determined by the length of the instruction look ahead.

4. SIMULATION OF ROUTERS

4.1 General Design

Since the Illiac IV and Semmelhaack routers have a similar structure, essentially the same program was used to simulate both routers. The sense in which the Illiac IV and Semmelhaack routers are similar is that both networks shift all data currently being routed in the network a certain fixed distance each routing cycle. The sequence and distances of the shifts are determined by some control hardware. Thus, a different subroutine is called from the same program to simulate the control of the Illiac IV or Semmelhaack router. Indeed, any router which has the characteristic of shifting all data a uniform distance during a certain routing cycle could be simulated with this program. One would only be required to build the proper subroutine to simulate the control network for the particular router. Since conflicts cause queue lengths and time through an Illiac IV type router for a set of input data vary depending on the input data, input frequency and control network simulated, the structure of this type router is simulated.

No hardware structure was simulated in either the crossbar switch or the Batcher network. As a result, the same program is used to simulate either the crossbar switch or a Batcher network. The simulation of the hardware structure of these two routers was not necessary. When a set of data to be permuted is introduced into either of these routers, the design of the

hardware assures that each data element will reach its destination after a fixed amount of time depending on the size of the router. Since our simulator is only concerned with a timing study, the program to simulate the crossbar switch and Batcher network merely delays a data element the proper amount of time before the element is placed in its destination buffer.

The third simulation program for routers is for the log router. Suppose the log router is required to accept 2^N data elements during a given routing cycle. In other words, the router width is 2^N. Then each of the data elements can be shifted distances of 0, 1, 2, 4, . . ., 2^{N-1} during any particular routing cycle. Notice the difference between the structure of the log router and the Illiac IV type. In the Illiac IV type router, all data being routed during a particular cycle must be moved a fixed distance; however, the log router design permits each of the data elements being routed to be shifted any of the allowed distances during any routing cycle provided the destination buffer is not full. The routing pattern in the log router is simply a binary number of length N set to the distance to be routed. A set bit in this binary number then indicates that a distance equal to the bits place value in the binary number should be included in the routing pattern. Since in this algorithm more than one element of data may arrive at a certain destination during a routing cycle, queues may form depending on the routing pattern of the input data and the frequency at which a new set of data is introduced into the

router. Thus, it was necessary to simulate the structure of the log router.

The general flow of each of the programs which simulate routers is shown in Figure 13. Since any of the routers can be placed in any of the router positions, a general interface program named DØNE is called during each routing cycle to empty buffers of elements which have reached their destination. DØNE empties routed elements into the proper memory or processor buffer depending on which simulation program calls DØNE and the current machine configuration. This program corresponds to the first block in Figure 13.

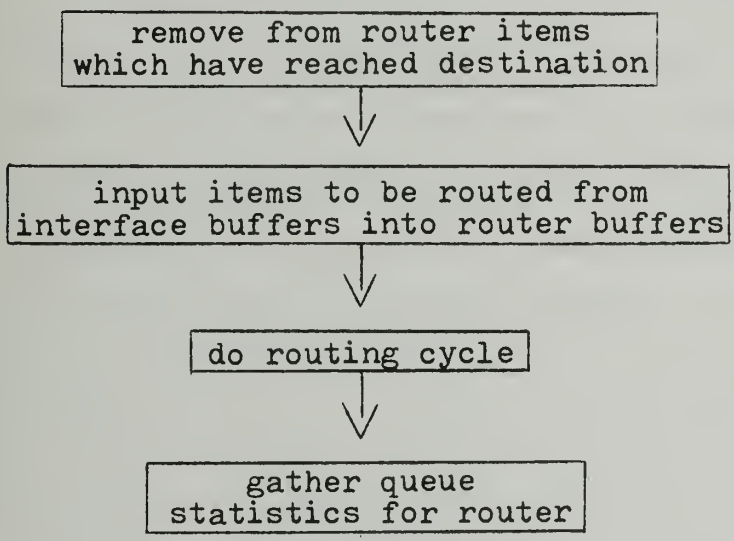


Figure 13

Two other interface programs are used. MBFTCH removes data from the memory buffers and places it into the interface buffers of R2. PDØNE is called when a result emerges from a

processor. PDØNE then places the data element in the proper interface router buffer depending on the current machine configuration.

4.2 Illiac IV and Semmelhaack Routers

Since queues form in the Illiac IV - Semmelhaack type router network as well as in the log router, some preliminary router simulations were done to determine what discipline for removing items from the router queues would result in the best throughput. It was found that the average time for each data element through a router, average queue length in the router and maximum number in any queue were minimum if the oldest data element relative to the time the elements were placed into the router have the highest priority to be shifted from each queue. Thus, for any router in which queues form, the router simulator removes data elements corresponding to the oldest machine instructions first.

Since the control program for the Illiac IV - Semmelhaack router is a subroutine, any sequence of shifts is possible. For the Illiac IV router shift distances of ± 1 and $\pm \lceil \sqrt{N} \rceil$ were simulated. N is the width of the router and $\lceil a \rceil$ denotes the next highest integer of a. For the Semmelhaack router, any distance uniform shift can be simulated. The sequence and distance of shifts used will be given in the results of each experiment in which either or both of the Illiav IV or Semmelhaack routers were used.

4.3 Crossbar and Batcher Networks

As mentioned previously, the program which simulates the crossbar-Batcher network type of router (or fixed time router) only delays each data element the time which the hardware would delay the data. If queues form at the input buffers of the router, the oldest data elements are removed first as in the Illiac IV-Semmelhaack case.

4.4 Log Router

The sequence of shifts for a given data element in the log router is fixed. The shifts are of distances 2^{N-1}, 2^{N-2}, . . ., 1 where N is the router width. Each element is shifted the highest possible power of 2 first, the next highest power of 2 next etc. It is clear that at most $\log_2 N$ shifts are required for each data element to reach its destination. As in the previous two simulation programs for the other two router types, the data element in the buffer corresponding to the oldest instruction in the machine is removed first.

5. EXPERIMENTS AND RESULTS

5.1 Discussion of Experiments

Unfortunately, time did not permit the performance of as many experiments as originally was hoped. Three different experiments were run on the simulator with the number of processors equal to seven. Then two of these experiments were simulated using fifteen processors. Each of these simulations used a three router configuration with all routers being the log router. In addition, some other simulations were completed to arrive at a relatively firm idea of what would be required in a system for the type of experiments performed in order that bottlenecks would not occur or would be kept at a minimum. For the additional experiments, router buffer statistics were not kept.

The FORTRAN program segments chosen to be simulated included a series of assignment statements taken from ACM FORTRAN algorithms. The other two experiments were back substituted sequences of assignment and conditional statements from ACM FORTRAN algorithms. Let some justification be given for this choice and the term "back substituted sequences" be clarified.

Programs in a high level language could be broken into parts consisting of iterative loops, sequences of assignment statements and sequences of conditional statements. An array machine like Illiac IV performs excellently on iterative calculations when arrays are being used. However, most of the

processors in this type of machine must remain idle when an assignment statement or the arithmetic involved in a conditional statement is being performed. In other words, sequences of assignment and conditional statements are potential bottlenecks in an array machine like Illiac IV. Obviously, the organization suggested here attempts to reduce this problem while keeping the array processing power. The array processing power could be kept by allowing router 2 to route data to all processors in the tree--not just the processors in the bottom level of the tree. In this way, the straight array calculation power remains. Since bursts of assignment statements and a mixture of assignment statements with conditional statements represent the major source of bottlenecks for a parallel processor, experiments involving these types of sequences were chosen.

In almost every nontrivial program, there exists at least one sequence of assignment and conditional statements. Since the tree processor can readily compute large expressions, it is reasonable to make a sequence of FORTRAN expressions into fewer and larger expressions to fill a processor tree. This can be accomplished by "back substituting" the expression for the assignment of a variable for any occurrence of that variable in succeeding statements of the sequence. This substitution can also be made for the arithmetic involved in any following conditional statements. In some cases the calculations involved for all paths through a sequence of statements could be done in as small or shorter time by a tree processor of a given size than

a single processor taking a single path. An example is shown of this type of back substitution in Figure 14. The FORTRAN sequence is shown in 14a and the back substituted sequence in 14b where

```
      A = B+C-D                    T1 = B+C-D-B-C+D-E-F
      R = A+E+F                    S1 = B+C-D-B-C+D+E+F+W
      IF (A.GR.R) GO TO 6          S2 = B+C-D+B+C-D+E+F-W
      S = A-R+W
      GO TO 7
    6 S = A+R-W
    7   .
        .
        .
```

a. b.

Figure 14

T1 represents the value of the arithmetic performed in the conditional statement. A single processor doing an add/subtract in 1 clock would require 7 clocks for one of the paths while a seven element tree processor would perform the back substituted conditional and assignment arithmetic in 5 clocks. After the paths were evaluated at run time, the effects of only the actual path taken would result, depending on the temporary values assigned for the conditional statements. In the example above, either S1 or S2 would be stored to memory location S depending on the value of T1.

5.2 Discussion of Results

The gathered statistics for each experiment are shown on Tables 1 and 2. Table 1 shows the time statistics for each experiment and Table 2 shows the processor utilization and router statistics (when taken) for the same experiments. Dashes (-) in the tables indicate that this statistic was not collected for the experiment. The description of the data for each experiment will be given below. The letters in the descriptions below correspond to the letters in the column marked "Data Set" in Tables 1 and 2.

A - Several sequences of assignment statements from ACM FORTRAN algorithms consisting of one hundred eighteen FORTRAN statements.

B - All paths through a sequence of assignment and conditional statements from the FORTRAN algorithm ADIPZ[6]. Each path was back substituted and the data contained seventy-nine FORTRAN expressions.

C - Same as B, except for the FORTRAN algorithm CHSTEP[6]. This experiment had ninety-six statements of which twenty-seven were of the simple assignment statement type (A=B).

D - This data consisted of the longest path from experiment C. This data was not back substituted and was eighteen FORTRAN instructions in length.

All time statistics are averages except where noted and represent time for the data for an entire instruction not individual data elements. Time for a certain statistic began elapsing when the first data element of an instruction reached the stage corresponding to the statistic being taken and was

terminated when the last data element of the instruction was removed from this stage. A description of the statistic represented by each of the columns in Tables 1 and 2 follows:

Data Set: References the experimental data used in the simulator. The code letters correspond to the descriptions of the data sets given above.

Number of PES: Number of processing elements in the tree for this simulation.

Number of Instructions: Number of processor tree instructions. In other words, the number of expressions generated for the simulation by the scheduler.

Instruction memory time: Average time from the clock that instruction enters the machine to the clock that all its associated data is available in a memory buffer.

Time in memory buffer: Average time that data for an instruction waits in memory buffer after available.

Time in Router 2: Average time data for an instruction spends in Router 2.

Time in PE tree: Average time data for an instruction spends in processor tree. (Time for whole expression to be evaluated.)

Time in PE buffer: Average time that computed result of an expression waits in output buffer of processor tree.

Time from PE buffer to memory: Average time that computed result spends in router when being routed from output PE buffer to memory.

Time from PE buffer to PE buffer: Average time that feedback result spends in Router 1 when being routed from output PE buffer to input PE buffer in bottom level of processor tree.

Total time of instruction: Average time from the clock that instruction enters the machine to

the clock that all processing and routing of results is completed.

Total clocks: Total number of clocks for entire experiment.

Max # in PE buffer: Maximum number of data elements at any one clock in a single processor buffer.

PE utilization #1 for adder: (Total number of results leaving all adders)/((Number of total clocks) * (Number of adders))

PE utilization #1 for multiplier: Same as expression for PE utilization #1 for adder except replace adder by multiplier.

PE utilization #2 for adder: (Total number of results leaving all adders from the clock at which the number of data elements in the processors was greater or equal to the number of adders to the clock when last instruction entered the machine)/((Number of clocks for which statistic taken) * (Number of adders))

PE utilization #2 for multiplier: Same as expression for PE utilization #2 for adder except replace adder by multiplier.

Ave # in R1 buffer: (Total number of elements entering R1 buffers)/((Number of R1 buffers) * (Total clocks))

Ave # in R2 buffer: Same as expression for Ave # in R1 buffer except replace R1 by R2.

Ave # in R3 buffer: Same as expression for Ave # in R1 buffer except replace R1 by R3.

Max # in R1 buffer: Maximum number of data elements at any one clock in a single buffer of R1.

Max # in R2 buffer: Maximum number of data elements at any one clock in a single buffer of R2.

Max # in R3 buffer: Maximum number of data elements at any one clock in a single buffer of R3.

Data Set	Number of PES	Number of instructions	Instruction memory time	Time in memory buffer	Time in Router 2	Time in PE tree	Time in PE buffer	Time from PE buffer to memory	Time from PE buffer to PE buffer	Total time of instruction	Total clocks	Max # in PE buffer
A	7	107	18.4	0.00	12.0	10.2	1.22	2.98	1.80	43.5	128	10
A	15	17	15.9	0.00	17.1	13.7	0.65	3.11	4.10	42.5	80	9
A	63	9	11.6	0.00	22.4	10.3	1.22	6.78	10.5	47.0	92	10
B	7	150	54.2	0.00	50.7	10.9	0.11	2.64	2.80	73.2	720	12
B	15	103	47.9	0.00	23.7	14.6	0.24	3.76	4.18	73.6	394	10
B	63	74	43.1	0.00	42.4	11.9	0.37	9.17	13.5	78.6	413	6
C	7	180	43.5	0.00	40.1	13.3	0.05	2.49	3.07	66.4	416	15
C	15	136	39.2	0.00	44.3	17.2	0.11	3.37	4.20	65.6	321	14
C	31	120	42.7	0.00	49.8	13.6	0.14	5.91	6.57	70.1	310	13
D	7	25	24.7	0.00	12.0	10.1	1.22	2.98	1.80	43.5	128	10
D	63	14	0.44	0.00	22.4	10.3	1.22	6.98	10.5	47.0	92	3

Table 1

Data Set	Number of PES	PE utilization #1 for adder	PE utilization #1 for multiplier	PE utilization #2 for adder	PE utilization #2 for multiplier	Ave # in R1 buffer	Ave # in R2 buffer	Ave # in R3 buffer	Max # in R1 buffer	Max # in R2 buffer	Max # in R3 buffer	Max # in memory buffer
A	7	0.08	0.09	0.09	0.14	0.56	2.98	0.49	3	4	3	15
A	15	0.04	0.05	-	-	0.06	0.46	0.03	2	4	1	11
A	63	0.01	0.01	-	-	-	-	-	-	-	-	10
B	7	0.05	0.05	0.05	0.05	0.05	0.27	0.03	2	4	2	24
B	15	0.04	0.04	0.03	0.03	0.05	0.36	0.05	1	4	2	23
B	63	0.01	0.01	-	-	-	-	-	-	-	-	23
C	7	0.09	0.10	0.10	0.11	-	-	-	-	-	-	24
C	15	0.06	0.06	0.07	0.08	-	-	-	-	-	-	24
C	31	0.03	0.03	0.07	0.09	-	-	-	-	-	-	23
D	7	0.08	0.10	-	-	0.54	2.90	0.32	2	4	2	16
D	63	0.01	0.01	-	-	-	-	-	-	-	-	3

Table 2

Max # in memory buffer: Maximum number of data elements at any one clock in a single memory buffer.

For all experiments, the maximum length of each router buffer was five. The adder pipeline was of length two and the multiplier pipeline was of length three. The experiments were all simulated assuming the instruction look ahead unit was of length ten. Up to fifty instructions could be simultaneously active in the simulator. The maximum memory buffer length was twenty-five and the maximum processor buffer length was twenty-two. For each experiment, one instruction was introduced into the simulator at each clock if the instruction look ahead unit was not full or the number of active instructions did not exceed the maximum.

Since only one machine configuration was simulated, no conclusion can be made concerning the best type of routers for each routing position. However, a few interesting observations can be made about the obtained results. The low average percent utilization of the processing elements obviously indicates a bottleneck at some location. The bottleneck appears to be that the memory modules supply Router 2 (hence, the processors) with data at too slow a rate. Apparently more than one data element for several instructions were stored in the same memory modules causing memory conflicts. This is reflected by the values in the column marked "Instruction memory time" in Table 1; most of the instructions required 30 to 40 clocks for all of the data to

be available from memory. Router 2 was clearly not the bottleneck for these experiments since the time data elements waited in memory buffers to be placed in Router 2 was 0. In addition, the average number of data elements in the buffers of Router 2 was small. Recall that the time in Router 2 is the time from the clock at which the first data element associated with an instruction arrives at the router until the clock at which the last data element of the instruction is removed. In determining the bottleneck, the statistic for max # in the memory buffers is misleading. For programming ease, the data for each instruction was placed in the memory buffers when the instruction was placed in the machine with a tag indicating at what clock the data could be removed. Thus, although the buffer size grew in the simulation, the hardware would not be required for such a large buffer. The gathering of this statistic should be changed to reflect the actual buffer size needed by the hardware.

To verify that the memory conflicts were causing the bottleneck, simulations were run which cycled the memory and routers more often than the processor tree. When the memory and routers were cycled three times as often as the processor tree, the number of total clocks for each experiment decreased drastically. The decreases in total clocks were as much as by a factor of between 4 and 5. When the memories and routers were cycled four times as often as the processor tree, Router 2 became the bottleneck. Another alternative to

release the bottleneck of memory conflicts is to map data to memory modules for instructions so that memory conflicts do not occur. This mapping could require the multiple storage of some variables.

Since Router 1 and Router 3 never became busy, a two router system probably would be more practical. The average number of data elements in the buffers of Routers 1 and 3 never reached 0.6. Thus, it would seem reasonable for a single router to route all results--results to memory and feedback results to the bottom level processing elements.

Finally, notice that merely increasing the size of the processor tree does not always reduce the total number of clocks necessary to complete the processing of a particular data set. The total time required by data sets A and B to be computed using 63 processors was greater than with 15 processors. The cause for this phenomenon is interesting. When no expression needs to be cut apart to fit on the current tree size, any larger processor tree would be too large for any expression in the sense that no operation would ever be carried out in the top level processor in the tree. In addition, the trees generated by the scheduler for the larger tree when more than one instruction is mapped onto the tree would be less dense. As a result, the average PE utilization would decrease. However, these sparse trees by themselves do not cause an increase in the total time taken for a data set to be computed. The increase is caused by the fact that routing distances in the router are longer which

causes an increase in routing time. Hence, one would suspect that no decrease in total time for an experiment would occur by increasing the number of processors when the largest expression fits on the current processor tree.

More simulations need to be completed to determine the adequate minimum router, processor and memory buffer lengths. Different types of routers should be attempted in various router positions. Especially interesting would be using a fast router like the crossbar switch as the result router. Without a great deal of work, the system could be changed to enable Router 2 to route data to all processors in the tree; complete FORTRAN programs including iterative loops could then be simulated with a more complex scheduler.

LIST OF REFERENCES

1. Kuck, D. J., Private conference.

2. Han, Joseph Ching-Chi, "Tree Height Reduction for Parallel Processing of Blocks of Fortran Assignment Statements", Master's Thesis, University of Illinois, 1971.

3. Muraoka, Yoichi, "Parallelism Exposure in Exploitation in Programs", Doctoral Dissertation, University of Illinois, 1970.

4. Budnik, Paul, "A Thesis Proposal", A paper written for preliminary examination at the University of Illinois.

5. Batcher, K. E., "Sorting Networks and Their Applications", Spring Joint Computer Conference, 1968, pp. 308-313.

6. University of Illinois, Department of Computer Science, "Subroutine Library".

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-72-503	2.	3. Recipient's Accession No.
4. Title and Subtitle Simulation of a Tree Processor			5. Report Date March 1972 6.
7. Author(s) Larry Allen Swanson			8. Performing Organization Rept. No.
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No US NSF GJ 27446
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C.			13. Type of Report & Period Covered M.S. Thesis 14.

15. Supplementary Notes

16. Abstracts

Recent trends in computer organization have tended to include more than one processor and more than one memory. In the future it seems likely that to achieve fast speeds many processors and memories will be used. A uniform way of thinking about such machine organizations is to assume that there are no direct connections between pairs of processors, pairs of memories or memories and processors. Rather all connections could be made through alignment networks or several alignment networks operating simultaneously. This thesis discusses the simulation of the operation of several such networks.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release Unlimited	UNCLASSIFIED	43
	20. Security Class (This Page) UNCLASSIFIED	22. Price